Distr.
LIMITED

UNIDO/IS.446
22 February 1984

UNITED NATIONS
INDUSTRIAL DEVELOPMENT ORGANIZATION

ENGLISH

SOFTWARE ENGINEERING: A SURVEY

by

Wladyslaw M. Turski**
UNIDO Consultant*

---

* The views expressed in this paper are those of the author and do not necessarily reflect the views of the secretariat of UNIDO.
Mention of firm names and commercial products does not imply the endorsement of UNIDO.
This document has been reproduced without formal editing.

** Professor, Institute of Informatics, Warsaw University, and Department of Computing, Imperial College, London.

V.84-82124

CONTENTS

	Page
The change in approach	2
Specification-based methods of programming	3
Approaches to specification of software	5
Software life-cycle	7
Modular design	9
Software modification	11
Software tools	13
Non-procedural languages and other advanced concepts in programming	16
Software engineering management	17

1. The term "software engineering" was coined at the same time and at the same conference which brought into the open the deep concern with the growing software crisis. The conference was held in 1968. The chief symptoms of the software crisis were: software unreliability, delays in meeting promised delivery time for software systems, difficulties in achieving desired functionality and performance of software, complexity of software systems and their resistance to modification attempts, shortage of skilled programmers and - above all - alarming cost of software development and maintenance.

2. The last fifteen years saw a dramatic decrease in the cost of all imaginable units of raw computing power: the dollar-per-KB and dollar-per-MIPS measures are falling in absolute terms, let alone discounted for inflation. Parallel to this trend, although running in the opposite direction, is a very rapid increase in hardware capabilities and availability. A fixed amount of money buys not only much more hardware today than 15 years ago, it also buys a much more sophisticated equipment. Software difficulties, then seen as a bottleneck in computer applications, have become the most important limiting factor today. For all practical purposes , the software costs are the foremost consideration when a new application system is contemplated. The policy of buying hardware to run available software systems has become almost the industry rule. The accumulated investment in software is already staggering and grows world-wide by some 10^{10} - 10^{11} US$ per year.

3. It should, therefore, come as no surprise that a very considerable effort is being put into improved methods of software design, implementation and maintenance. Qualified programmers being scarce the world over, assorted software tools, increasing the programmers productivity by automation of the more routine aspects of their work, are considered a very promising means of alleviating the software crisis.

4. In this report we shall survey the main directions of software engineering - a discipline of producing better software more economically. (It is important to consider both aspects simultaneously: making software cheaper at the expense of its quality is just as absurd as improving its quality at a disproportionate increase in costs).

The change in approach

5. Arguably the most important change in the whole software scene over the last 15 years is the emergence of consensus on the issue of software correctness. It is by now universally accepted that correctness is the main criterion of software quality: no matter how good a piece of software is in all other respects (such as efficiency or robustness), if it is incorrect its value is nil. Without an accepted notion of what constitutes the correctness of software, the insistence on software being first of all "correct" would be meaningless.

6. A very useful notion of software correctness has been found in the logical notion of satisfaction that may exist between two formal systems. Roughly speaking, a system S satisfies system T if whatever follows from system T is a fact in system S. In more rigorous terms, we say that S satisfies T if there exists an interpretation I: T - S such that to each statement t deemed true in T the interpretation I assigns a provably true statement s in S: (t = true in T) implies (s = I(t) = true in S). In practical terms, T is what constitutes a specification, S - the software.

7. For example, statement t may say that SORT(x) is the sequence X rearranged so that its members are put in ascending order. Statement s may take a form CALL PROC SHELL(INPUT(A)). If now the interpretation I is such that PROC SHELL is the name of the Shell-sort routine, CALL denotes an invocation of a routine, A is the name of a file, INPUT - an operation delivering the file listed as its parameter, then all that is needed to establish that s satisfies t is to prove - from the particular description of Shell-sort - that indeed its execution delivers the sorted version of its input parameter.

8. Two observations are in order:

(a) Using the outlined approach we assume that the specification correctly reflects the user's requirements.

(b) It is a matter of formal calculations to establish if a given software satisfies the given specification.

9. The first of these observations clearly indicates that the burden of somehow verifying whether or not the software meets the application needs is shifted from programming to specification analysis. The second one presents in a nutshell the methodological advance which is the cornerstone of software engineering: given the specifications, the correctness of remaining parts of the design and implementation process becomes a calculable question.

10. We shall return to the specification issue later. Now, we shall consider the significance of correctness calculability.

11. First attempts to exploit the notion of calculable correctness concentrated around programme verification, i.e. around a process which would take a specification and a programme, and - based on this information alone - would attempt to calculate if the programme is correct. This approach has had a limited success only: the amount of formal calculations involved was formidable even in the case of pretty small programmes. For large programmes it became prohibitive, even if human intervention was allowed to speed up some of the calculations.

12. Soon it transpired that a much better policy to exploit the notion of calculable correctness is to devise such programming techniques which would guarantee programme correctness by virtue of the very construction process. Thus methods of building correct programmes from specifications started to appear.

Specification-based methods of programming

13. Common to these methods is the view of specification as the most abstract description of all desired properties of the desired software. (By "most abstract" here we mean "free of all unnecessary detail".) The software design and implementation is seen as a process of transforming such abstract description, by adding necessary details, into a programme which is to preserve all properties contained in the specification. (Thus the main difference between the specification and the corresponding programme is that of detail: the specification is free of machine-oriented details but, of course, contains all application-oriented ones.)

14. Methods which pursue this approach are known as top-down design methodologies, a name which refers to the fact that - after the software is successfully designed and implemented - the history of the construction process is not unlike a pyramid with the original specification occupying its summit and the working version of the implemented software being its base. When one ascends this pyramid (moving as it were in the direction opposite to that which the designer took) one sheds the implementation and design details until the refined, most abstract summit - the specification - is reached.

15. Basically, the top-down programming methodology consists in repeated application of the following procedure:

(a) Given a problem P, is it possible to express its solution in a reasonably concise fashion using primitive notions of the linguistic level at which we want to programme? If yes, write the programme, if not, invent notions P1, ..., Pn such that

(i) each of the notions P1, ..., Pn is well-specified,

(ii) using these notions according to their specification it is possible to write a satisfactory programme for problem P.

(b) Consider each of the notions P1, ..., Pn in turn as a new problem and repeat the procedure.

This problem continues until all invented notions are implemented in terms of primitives of the given programming level.

16. The above given brief description of the top-down design and implementation methodology introduces two important techniques: that of structured programming (or structured decomposition) and that of stepwise refinement. We rely on the first one when we decompose problem P into problems P1, ..., Pn, and on the other one when we consider each of the P1,..., Pn as a new problem in itself, to be solved by the same method.

17. Both techniques rely on sound mathematical principles which guarantee their correctness if certain rules of decomposition and refinement are observed. Consideration of the mathematical foundations of structured programming and stepwise refinement have led to new concepts in programming languages, such

as ADA, PASCAL or MODULA. These modern programming languages are designed explicitly so as to facilitate, and even in some instances: enforce a disciplined use of these techniques. It should be observed that while no programming language per se ever solves any software design problem, the use of a tool influences the way in which its user works. Programming languages of yesteryears, FORTRAN, COBOL and BASIC lack the mechanisms to support structured programming and stepwise refinement. In such languages it is virtually impossible and certainly very awkward to use the techniques which emerged as the most common tool of modern software engineering.

Approaches to specification of software

18. It has been firmly established that the early stages of software system design are crucial for the eventual system usefulness. This observation is, of course, directly related to the fact that it is precisely the specification which in the final count is taken as the frame of reference in which software correctness is established. Thus, all design/implementation techniques respecting the notions of correctness cannot but preserve any specification errors. Consequently, such errors become apparent only after the software has been implemented and thus are very expensive to correct. This clearly underscores the need to verify specifications, both from the point of view of their implementability and from the point of view of their relevance for the intended application.

19. The verification of specifications with respect to their relevance poses a very subtle question of translation between (often fuzzy) intentions of the eventual user and (necessarliy formal) expression of specifications. Many techniques have been proposed specifically oriented towards easing of this task. In essence, these techniques assume that the prime author of the specifications is the eventual user, and - by providing a somewhat restrictive means of expression - force him to express his intentions in a form that can be easily manipulated by formal methods. Often such techniques depend on graphic conventions (e.g. SADT), whereby the user/author is pressed to describe his ideas in form of pre-designed and partially labelled diagrams. Freely added, user-invented lables express his intentions originally just by their mnemonic significance. Gradually, as the specifier is asked to complete more

detailed diagrams, the pre-designed structural dependencies between diagrams are explored by a "hidden" analyser, which brings to the open all inconsistencies and many instances of design incompleteness.

20. An entirely different approach to specification writing can be exemplified specification languages (such as CLEAR). In this approach, the recognition that a specification is in fact a formal theory of an application domain is made into the main tenet and main mental tool as well. Specification languages provide means for relatively easy-to-read description of theories and - more significantly - for combining thus described theories into larger ones. For example, given a formulation of a theory of optimality (for instance by means of linear programming principles) and a formulation of a theory of control of a chemical process, their combination will yield a formulation of a theory of optimal control of this process. (Naturally, not any two theories can be combined, it is up to the specification language designers to make sure that any combination expressible in the language "makes sense" and that absurd combinations would be inexpressible; exactly as in safe programming language it is impossible to execute sin(true) instruction.)

21. When many useful application domain concepts are captured by corresponding theories, a specification language may be indeed very useful: the specification of a concrete system may be obtained by a combination of "library theories" with some specific expressions written just for this system. The main advantage of this approach, apart from economy of design, is in the increased safety: library theories are known to be safe and the language includes many safety measures which make it unlikely that the nonsensical combinations would be expressed by mistake.

22. Both graphic and linguistic approaches to formal specification writing are well founded in deep theoretical research into such issues as abstract data types, algebraic theories, theory of models etc. The same kind of foundations are used by a number of software description techniques, such as VDM, commonly used for unambiguous definition of large software products, such as semantics of new programming languages (CHILL, ADA) or special software systems (CICS).

Software life-cycle

23. A software system written for a particular application seldom can be used for an extended period without undergoing a number of changes. Among the many causes that necessitate software changes one can list the following:

(a) The nature of the application itself changes (e.g. for a banking application, the introduction of EFT facility changes dramatically the accounting procedures).

(b) New hardware elements are added to the system and need to be incorporated into the class of devices supported by the software (e.g. colour screens are introduced into a system that used monochromatic screens only: a new "dimension" must be added to all output and, perhaps, input functions).

(c) An existing piece of saftware is transferred to another environment, or the environment itself changes (e.g. better educated operators are hired, for whom the existing input procedures are too dull; new input procedures are required which would make the operators' task more appealing to new staff).

(d) The scope of the application is enlarged (e.g. in a hospital computer system the intensive-care unit computer services are to be linked to a previously separate medical record system).

(e) The requirements placed on the existing system are changed (e.g. the air-traffic control system must be modified when the airport it serves starts accepting faster jets and thus the decision time must be reduced to accommodate the faster traffic).

The list of causes can be, no doubt, extended. Even this incomplete list is sufficient to draw the unavoidable conclusion: changes in a live system are indeed necessary, quite apart from any remedial changes due to its detected stortcomings and internally motivated software improvements.

24. An unfortunate tradition lumps under the title "software maintenance" all activities related to software changes occurring after the original system has been successfully installed. It is important to remember therefore that software maintenance is needed not because software deteriorates in use

(there is, of course, no wear and tear of software), but because its use for a dynamically changing application will be diminished unless the software is modified.

25. In fact, the usual presentation of software life-cycle contains four major phases:

(a) Conception
(b) Design
(c) Implementation
(d) Maintenance.

26. Largely due to the fact that the "maintenance" comprises future modifications, its share in the total expenditure is very large. In many cases the maintenance costs constitute more than three-quarters of the total investment in a software system. (This is a very important observation: a client buying a moderately-sized software system for, say $ 100,000 should not be surprised that the maintenance costs over the next few years will run up to $ 300,000. If he is not prepared to pay this "extra", almost certainly he will find himself tied to an ever decreasing useful piece of software. If he tries to economize, e.g. by assigning junior staff to maintenance activities, he may face a total disaster - the system may become hopelessly entangled and virtually useless.)

27. The main reason for the high cost of software maintenance is the (abundantly confirmed by many a post-mortem analysis) fact that the complexity of software grows very rapidly with every change made to the original version, unless a conscious redesign efforts (also expensive) is made to reduce the complexity. Thus, each subsequent change is harder to make and is more likely to introduce - in addition to the desired ones - many unforseen and often inpleasant effects.

28. These observations lead to two inter-related software engineering problems:

(a) How to design software in such a way that it would be relatively easy to modify?

(b) How to modify an existing software so as not to increase its complexity more than absolutely necessary?

29. The first of these issues is answered by modular and hierarchical design. The second - by controlled backtracking techniques - greatly facilitated by well-designed programming support environments.

Modular design

30. A very general engineering principle calls for the final product to be assembled from easily replaceable parts. Thus a bicycle consists of a frame, two wheels, pedals, chain etc. If the bicycle malfunctions, the cause of the trouble can usually be traced to one of these parts, and the faulty part can be replaced by another of the same type or, indeed, by any similar part which fits. It is not unusual for a bicycle to have wheels of different make than the frame or even two different pedals. As long as the parts satisfy certain externally specified interface requirements, their internal construction is of secondary importance. Thanks to this principle, we may put snow tyres on our cars, thus obtaining a vehicle with rather different driving parameters, without actually having to change the engine or steering wheel.

31. This general engineering principle translates in software design into a requirement according to which any software product should be built from relatively independent modules. Each module meets its specifications if it is a correct module; a module specification is all that is externally known about the module. The functionality of the whole system obtains from the interaction of modules, the interaction itself being fully determined by modular interfaces.

32. When designing a piece of software, one decomposes the design into a number of relatively independent units - modules. Having listed the external properties of each module, i.e. having formulated each module specification, one can prove the appropriateness of the decomposition by proving that the required properties of the whole indeed follow from the postulated properties of modules. This being established, each module in turn may be considered a new designing problem. Thus, allowing for hierarchy of modules, we see a complete analogy with the stepwise refinement technique, although this time the technique is expressed in terms of structured components of programmes.

33. A well modularized programme can be now relatively easily changed by replacement of a module by another, the fresh module sharing with the old one its interface specification, while differing in secondary considerations, i.e. in those which are not covered by the specification relied upon in the overall design. For instance, if we want to adapt our software to exploit the potentials offered by a new output device, we can concentrate on the output module in which all relevant aspects of the given piece of software should be encapsulated.

34. Naturally enough, not any haphazard hacking of the design into pieces can be considered a proper modularization. Sound engineering principles of modular design have been formulated, which facilitates making correct modularization decisions. The same principles ensure that a majority of errors can be localized within a module, thus the repair activities may usually be limited to a single module.

35. Since a module can by replaced by any other module provided their functional specifications and interface characteristics match, there is a huge incentive to plan libraries of interchangeable modules, rather like a Mechano set, from which a variety of software products could be rapidly constructed. Modules that could be used in many different programmes are known as reusable ones, and all successful software houses possess substantial libraries of reusable modules from which a major part of any system within the specialization area of the house can be constructed.

36. Some modern programming languages, notably ADA and MODULA, actively encourage modular programming. ADA, for example, was explicitly designed to permit independent compilation of modules, so that libraries of reusable, precompiled modules bay be accumulated.

37. A number of design methods have been proposed which are based on the modular programming principle. Some of these are commercially available as kits consisting of a large number of tools, i.e. special programmes which assist programmers in their work on software design and implementation. In general, such commercially available methods concentrate on a chosen guiding principle for modularization (e.g. data flow, calling hierarchy or input/output

transformation) which suggests a particular approach to structuring the design. Identified modules are named and their main characteristics are specified. Then, the use of supplied tools enables one to display the emerging design in a coherent way and - more importantly - to check the consistency of the design by verifying that the specified properties of the modules placed in their respective structural positions indeed correspond to each other. Thus, for instance, if it follows from the design structure that a module M imports data named x, it is possible to check that there is at least one other module which exports thus named data. Having identified such a module, say N, it is possible to check if modules M and N are structurally related in such a way that data transfer between them is allowed. Similarly, it may be checked if data z, generated by a module K, is imported by any other module or presented as a system output.

38. Most available tool-assisted modularization methods allow many such consistency checks to be run at several levels of detail, thus permitting to verify at least some aspects of the design before any detailed programming (on intramodular level) is done.

39. Another useful extension of modular programming techniques consists in substituting module surrogates for not yet implemented modules. Thus, as the implementation progresses, one can animate the complete system even if only a part of its modules is actually coded: the remaining modules being replaced by surrogates, the whole system may be made to perform. Such animated execution allows to check some external properties of the system long before its implementation is completed and therefore to avoid at least some unpleasant surprises and - perhaps - to introduce design modifications before they become prohibitively expensive to carry out.

Software modification

40. Assuming that we have a well designed and correctly implemented piece of software, any subsequent modification should start with a clearly specified request for modification. If the history of the design is preserved (and again there are special software tools constructed specifically for the purpose of storing the software design history in a manageable form), it is

possible to identify the design step at which the decisions contrary to the requested modification were made. (Such a step must always be there since otherwise the request could be met by the existing software and thus no real change would be in fact requested although the request may still have led to some programming, e.g. of an add-on extension of the available software.) As soon as the pertinent design step is identified we know that all preceding steps can be safely preserved in the new design, i.e. in the design aimed at satisfying the considered change request. If the subsequent steps of the old design are discarded, the incorporation of the requested modification may be viewed as a continuation of the preserved part of old design by a suitable sequence of the new design steps.

41. If there are many such modification requests, we are soon faced with a "forest" of designs - from each design step at which some change has been incorporated a new branch is started. A suitable "navigation" tool is needed if the programmer is to be able to traverse freely the design "forest" and collect design steps along each particular branch. Again, such tools are commercially available.

42. The main objective of controlling the software modification may be stated as a stability problem for software development: how to achieve the situation in which small changes in specification could be accommodated by small changes in the implemented software.

43. No general solution of this problem is known (and, according to many experts, such general solution may never be discovered). There exist, however, some design/implementation techniques which admit a partial solution of the stability problem.

44. For instance, if every time an arbitrary decision is made (selecting a particular branch) all feasible although discarded decisions are duly recorded and preserved in the design data base, subsequent design steps may include a "what if" analysis, forcing the programmer not to do anything detrimental to implementation of the rejected options. or - at the very least - to clearly mark subsequent decisions with comments informing about such past options which from now on become infeasible. If the design history is decorated with such comments, a change request may be run against the tree of rejected

alternatives, yielding not only the level to which the design is to be backtracked, but also informing about the point at which the incorporation of the requested change became infeasible in the present implementation. Analysis of this information permits to guess roughly the amount of redesign effort needed to make the change. Changes that would require too much effort may be then rejected, or - if undertaken - may be explicitly marked as difficult-to-implement and therefore expensive. It should be stressed once more that a proper management of software modification and change requests is probably the most important aspect of software engineering as it is this phase of the software life-cycle in which the lion's share of the total expense is incurred. The general approach outlined above may be viable only if during its entire life-cycle a software system is supported by a comprehensive design and development documentation in which all versions and mutations are recorded in a manner allowing for a relatively easy restoration of arbitrary past states. It goes without saying that if such a support is to be of real assistance to programmers, it must provide the relevant information in a form that permits machine-assisted manipulation of a large number of variants. Thus again we are led to consider the importance of software tools.

Software tools

45. There are several varieties of software tools more or less available on the market, many more tools are the property of software houses which use them for their internal purposes.

46. Probably the most common among commercially available tools are all sorts of editors, i.e. programmes that facilitate programme composition and programme text manipulation at a programmer's work station (e.g. on a VDU). The editors range from pretty simple text manipulators to quite sophisticated, programming-language oriented structured editors that in addition to the facilities provided by plain text editors include an active mode of assistance. In the latter mode, structured editors prompt the programmer as to proper instruction formats, check for syntactic completeness of phrases, warn against simple context-detectible errors etc. Nearly all varieties of programming editors take care of simple yet laborious editing operations, such as textual substitutions, systematic renaming of programming objects etc. Many newer programming editors are geared to exploit display facilities offered by modern

terminals, e.g. by providing the so-called multiwindow option, whereby a programmer may divide the screen into several independent "windows", conducting in each of them a separate programme (or programme part) development. Thus, for example, a programmer may develop the main programme in one window, a subrouting in another and an input/output handler in yet another. A fourth window may be used for display of pertinent parameters, such as the number of lines of code already generated, memory maps etc. Each of the windows may be zoomed in, therefore providing a more detailed view of a particular feature, contents of different windows may be merged, etc. A programming editor with multiwindow facility creates a fair analogy of programmer's desk top, with all assorted documents and scratch pads being available in an electronic form at the same time.

47. Another kind of commonly used programming tools is represented by programme generators. For a variety of applications, working programmes are sufficiently stereotyped to permit their automatic generation from a suitable chosen set of design parameters. Such programme generators usually work interactively, either in a dialogue (question and answer) mode or by menu selection technique, whereby a programmer is shown a number of options, depending on the circumstances selects one, thus fixing a design decision, which triggers a next level menu to be displayed. At the end of a session, the totality of decisions made determines a particular application programme which is then produced ready for operation. Programmes produced in this way are often a bit inefficient, but otherwise quite acceptable and certainly can be used as system prototypes. If their functional behaviour is found to be satisfactory, their performance may be improved in a number of ways, e.g. by optimizing the most frequently executed parts of the code. (Incidentally, programme optimization is another task that often may be left to a suitable software tool.)

48. Finally, we should not forget that he ever growing body of commercially available compilers, interpreters, decision-table processors etc. are all in fact software tools. An extremely useful addition to this class of tools are programme transformation systems which accept a programme expressed in an abstract form and - guided by the programmer - perform textual transformations aimed at replacement of abstract algorithms by concrete ones, or at replacement of less efficient parts of code by more economic versions. The importance

of such tools rests in the fact that programme transformation systems are so designed that their action preserves the intended meaning of programmes being transformed. A programmer may therefore try a number of transformations quite safely: even if he does not achieve the intended improvement, he certainly does not run a risk of losing correctness of his programme.

49. Another class of software tools are those which gradually transform an initial design into a more and more programme-like text. Some tools of this class accept, for instance, graphic designs in form of interconnected, labelled boxes and represent them as equivalent linguistic structures more amenable to further textual refinement. (In addition to transformations between various levels of abstraction, such systems usually perform a number of useful consistency checks.) Nearly all software design methodologies advocated for general use rely on some tools of this class.

50. The most sophisticated software development tools, in addition to all previously listed facilities and standard features (such as parsers, table generators and file managers), incorporate also very advanced data bases in which programme versions and mutations are stored for easy reference and manipulation.

51. It is customary to refer to a fully developped system of software support tools as programme support environment (PSE). Modern requirements for a programming language usually include a specification for a PSE. Probabaly the best-known of them is the APSE - intended programming support environment for ADA. In fact, it is expected that the portability of ADA programme will be achieved via functional equivalent of APSE installed at nearly all computers. In this way, not only the programmes themsevels could be ported, but also their PSE, which would allow for a further development of a programme to be ported from one installation to another.

52. In addition to programme design/implementation tools, a well-developed PSE includes a variety of tools for programme testing and debugging which greatly reduce time and effort needed for the pre-release servicing of software.

Non-procedural languages and other advanced concepts in programming

53. Classic programming languages, COBOL, FORTRAN, PASCAL and even ADA, are based on the notion of assignment: as a result of the execution of a statement, a variable is assigned a new value. This notion, a direct descendant of machine order "execute and store", can be rightly described as the cornerstone of the von Neumann computer architecture.

54. New architectural concepts, such as data-flow machines, highly parallel computers and inference engines, find little use for the notion of assignment. Hence, in recent years a number of entirely different programming languages have been proposed, based on very unorthodox principles and incorporating more or less directly these architectural innovations which seem most promising from the application point of view.

55. Arguably the most widely accepted of the new line of programming languages is PROLOG, a language for programming in logic. Originally intended as a tool for computational linguistics, PROLOG is rapidly becoming the main programming language for artificial intelligence, slowly replacing in this role the old-time favourite, LISP.

56. What an assignment statement is to FORTRAN or PASCAL, a Horn clause is to PROLOG. (A Horn clause is a special form of a first order predicate calculus formula.) Its main computational advantage rests in the natural parallelism of Horn clause evaluation, while its advantage for application programming obtains from the observation that a Horn clause may be just as easily interpreted as a statement of a fact and as statement of a hypothesis to be verified based on other clauses. Close kinship of Horn clauses and relations which provide the foundations for commonly used relational data bases is another bonus for PROLOG adherents: not only does it yield an easy interface between application programmes and data bases, but also makes possible data base description by essentially same means as programme description.

57. Data flow machines seem to provide a natural evaluation mechanism for another brand of new programming languages, the so-called applicative languages, exemplified by LUCID. Hardware/software experimental systems, e.g. ALICE, are currently being built to explore new and apparently very powerful concepts of data flow and functional programming.

58. Finally, the Japanese Fifth Generation project, extremely influential in shaping the current research interests both in the USA and in Europe, favours highly parallel machine architecture harnessed to a multilevel hierarchy of specialized machines: data base machine topped with an inference engine (programmed in PROLOG) yields an expert knowledge base which, interfaced with very advanced input-output machines (such as graphic or visual computers and speech analysers and synthesisers), are to become intelligent computers of the 1990s.

59. Futuristic as such designs may seem, they do underline a new approach to computing: a merger of hardware and software design, aimed at exploitation of the potentials made available by the abundance of cheap and very efficient large chip components.

Software engineering management

60. As soon as we recognize that software writing has become the major component of the multi-billion dollar information processing industry, we must realize that this kind of industry generates its peculiar managerial problems and leads to specific managerial techniques.

61. The peculiarities of the software industry are quite pronounced. First of all, it is an industry almost totally independent of any raw materials and almost zero-energy consuming. Apart from software tools - themselves produced within software industry - it does not require much investment in material terms. Its products are often classified as intangibles, and - contrary to any other industry - most of its costs concentrate in design: the actual production (if one considers the reproduction of once composed programmes as "production") costs are practically nil. This should be contrasted not only with usual industrial sectors, such as mining or manufacturing, but also with innovation-intensive industries, such as civil engineering or drug industry. (A constructing firm may redesign a bridge ten times over, the cost of nine discarded designs is neglegible as compared with that of actually building a bridge on site. A software firm forced to redesign a major piece of software usually comes close to bankruptcy).

62. In addition, nearly all resources needed for the software industry are people: highly qualified, well educated software engineers. Hence, the managerial problems in this industry are almost exclusively pure problems of the workforce management, where the workforce in question is very independent and fully aware of its own value.

63. To fully understand the ensuing managerial difficulties, let us consider a simple example: a software team engaged to produce a system for an application, is running behind schedule. The manager decides to accelerate the rate of progress and implements this decision by hiring more programmers. Instead of expected acceleration, the work is slowed down as the fresh programmers need to be instructed about the system in production and the only available instructors are the programmers already on the job. Thus the newly hired hands are unproductive because they are not "in" yet, while the old hands, burdened with the additional task of instructing the new-comers, become less productive. Hiring even more programmers may be disastrous. After a period, the newcomers are integrated and the work may resume in earnest (although the delay has grown considerably). Now, however, the management discovers that the enlargement of the team has blown up the communication problems which grow as the square of the number of people on the project, and a sizeable portion of the total effort must be spent on overcoming the resultant communication clashes. Hiring an extra support team: secretaries, information officers and technical writers compounds the difficulty. After another couple of months the management realize that they are fighting an uphill battle: the effective rate of system production is invariant of the work expanded, it seems to be solely determined by the original design and after it has been approved there is little that the management can do to influence the rate of further development.

64. A number of managerial techniques, including the spectacularly successful "chief programmer team" approach pioneered by IBM, has been proposed and found useful. Still, the management of software projects remains a vexing problem and successful software managers are - if anything - even rarer than good programmers.